

# The binding model: Retrieving and validating user input

---

## ***This chapter covers***

- Using request values to create binding models
- Customizing the model-binding process
- Validating user input using `DataAnnotations` attributes

In chapter 5 I showed you how to define a route with parameters—perhaps for the day in a calendar or the unique ID for a product page. But say a user requests a given product page—what then? Similarly, what if the request includes data from a form, to change the name of the product, for example? How do you handle that request and access the values the user provided?

In the first half of this chapter, we'll look at using *binding models* to retrieve those parameters from the request so that you can use them in your Razor Pages. You'll see how to take the data posted in the form or in the URL and *bind* them to C# objects. These objects are passed to your Razor Page handlers as method parameters or are set as properties on your Razor Page `PageModel`. When your page handler executes, it can use these values to do something useful—return the correct diary entry or change a product's name, for instance.

Once your code is executing in a page handler method, you might be forgiven for thinking that you can happily use the binding model without any further thought. Hold on though, where did that data come from? From a user—you know they can't be trusted! The second half of the chapter focuses on how to make sure that the values provided by the user are valid and make sense for your app.

You can think of the binding models as the *input* to a Razor Page, taking the user's raw HTTP request and making it available to your code by populating "plain old CLR objects" (POCOs).<sup>1</sup> Once your page handler has run, you're all set up to use the *output* models in ASP.NET Core's implementation of MVC—the view models and API models. These are used to generate a response to the user's request. We'll cover them in chapters 7 and 9.

Before we go any further, let's recap the MVC design pattern and how binding models fit into ASP.NET Core.

## 6.1 Understanding the models in Razor Pages and MVC

In this section I describe how binding models fit into the MVC design pattern we covered in chapter 4. I describe the difference between binding models and the other "model" concepts in the MVC pattern, and how they're each used in ASP.NET Core.

MVC is all about the separation of concerns. The premise is that by isolating each aspect of your application to focus on a single responsibility, it reduces the interdependencies in your system. This separation makes it easier to make changes without affecting other parts of your application.

The classic MVC design pattern has three independent components:

- *Controller*—Calls methods on the model and selects a view
- *View*—Displays a representation of data that makes up the model
- *Model*—The data to display and the methods for updating itself

In this representation, there's only one model, the application model, which represents all the business logic for the application as well as how to update and modify its internal state. ASP.NET Core has multiple models, which takes the single responsibility principle one step further than some views of MVC.

In chapter 4 we looked at an example of a to-do list application that can show all the to-do items for a given category and username. With this application, you make a request to a URL that's routed using `todo/listcategory/{category}/{username}`. This returns a response showing all the relevant to-do items, as shown in figure 6.1.

The application uses the same MVC constructs you've already seen, such as routing to a Razor Page handler, as well as a number of different *models*. Figure 6.2 shows how a request to this application maps to the MVC design pattern and how it generates the final response, including additional details around the model binding and validation of the request.

---

<sup>1</sup> POCOs are regular C# classes and objects that live in memory. There are no special inheritance requirements or attributes required to use them with ASP.NET Core.

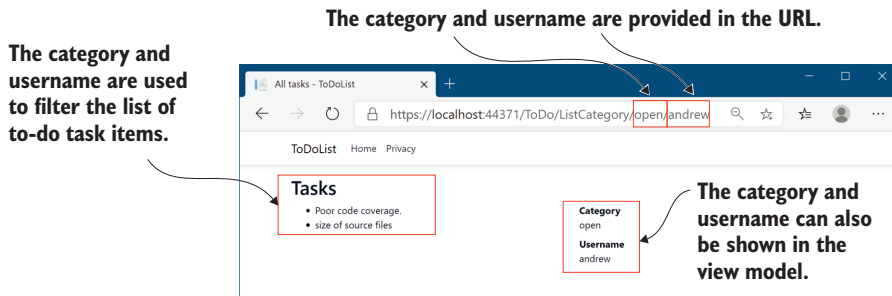


Figure 6.1 A basic to-do list application that displays to-do list items. A user can filter the list of items by changing the category and username parameters in the URL.

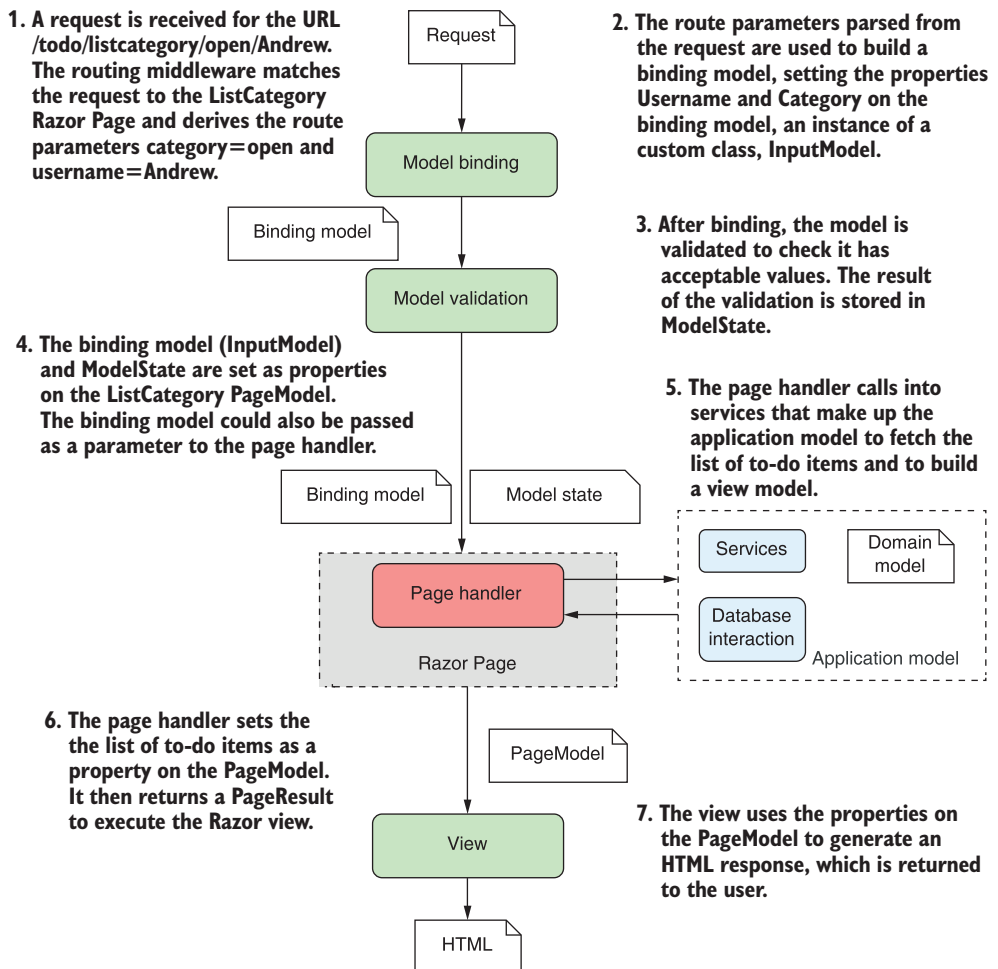


Figure 6.2 The MVC pattern in ASP.NET Core handling a request to view a subset of items in a to-do list Razor Pages application.

ASP.NET Core Razor Pages uses several different models, most of which are POCOs, and the application model, which is more of a concept around a collection of services. Each of the models in ASP.NET Core is responsible for handling a different aspect of the overall request:

- *Binding model*—The binding model is all the information that’s provided by the user when making a request, as well as additional contextual data. This includes things like route parameters parsed from the URL, the query string, and form or JSON data in the request body. The binding model itself is one or more POCO objects that you define. Binding models in Razor Pages are typically defined by creating a public property on the page’s `PageModel` and decorating it with the `[BindProperty]` attribute. They can also be passed to a page handler as parameters.

For this example, the binding model would include the name of the category, open, and the username, Andrew. The Razor Pages infrastructure inspects the binding model before the page handler executes to check whether the provided values are valid, though the page handler will execute even if they’re not, as you’ll see when we discuss validation in section 6.3.

- *Application model*—The application model isn’t really an ASP.NET Core model at all. It’s typically a whole group of different services and classes and is more of a concept—anything needed to perform some sort of business action in your application. It may include the domain model (which represents the thing your app is trying to describe) and database models (which represent the data stored in a database), as well as any other, additional services.

In the to-do list application, the application model would contain the complete list of to-do items, probably stored in a database, and would know how to find only those to-do items in the open category assigned to Andrew.

Modeling your domain is a huge topic, with many different possible approaches, so it’s outside the scope of this book, but we’ll touch briefly on creating database models in chapter 11.

- *Page model*—The `PageModel` of a Razor Page serves two main functions: it acts as the controller for the application by exposing page handler methods, and it acts as the view model for a Razor view. All the data required for the view to generate a response is exposed on the `PageModel`, such as the list of to-dos in the open category assigned to Andrew.

The `PageModel` base class that you derive your Razor Pages from contains various helper properties and methods. One of these, the `ModelState` property, contains the result of the model validation as a series of key-value pairs. You’ll learn more about validation and the `ModelState` property in section 6.3.

These models make up the bulk of any Razor Pages application, handling the input, business logic, and output of each page handler. Imagine you have an e-commerce

application that allows users to search for clothes by sending requests to the `/search/{query}` URL, where `{query}` holds their search term:

- *Binding model*—This would take the `{query}` route parameter from the URL and any values posted in the request body (maybe a sort order, or the number of items to show), and bind them to a C# class, which typically acts as a throw-away data transport class. This would be set as a property on the `PageModel` when the page handler is invoked.
- *Application model*—This is the services and classes that perform the logic. When invoked by the page handler, this model would load all the clothes that match the query, applying the necessary sorting and filters, and return the results to the controller.
- *Page model*—The values provided by the application model would be set as properties on the Razor Page's `PageModel`, along with other metadata, such as the total number of items available, or whether the user can currently check out. The Razor view would use this data to render the Razor view to HTML.

The important point about all these models is that their responsibilities are well defined and distinct. Keeping them separate and avoiding reuse helps to ensure your application stays agile and easy to update.

The obvious exception to this separation is the `PageModel`, as it is where the binding models and page handlers are defined, and it also holds the data required for rendering the view. Some people may consider the apparent lack of separation to be sacrilege, but in reality it's not generally an issue. The lines of demarcation are pretty apparent. So long as you don't try to, for example, invoke a page handler from inside a Razor view, you shouldn't run into any problems!

Now that you've been properly introduced to the various models in ASP.NET Core, it's time to focus on how to use them. This chapter looks at the binding models that are built from incoming requests—how are they created, and where do the values come from?

## 6.2 *From request to model: Making the request useful*

In this section you will learn

- How ASP.NET Core creates binding models from a request
- How to bind simple types, like `int` and `string`, as well as complex classes
- How to choose which parts of a request are used in the binding model

By now, you should be familiar with how ASP.NET Core handles a request by executing a page handler on a Razor Page. You've also already seen several page handlers, such as

```
public void OnPost(ProductModel product)
```

Page handlers are normal C# methods, so the ASP.NET Core framework needs to be able to call them in the usual way. When page handlers accept parameters as part of

their method signature, such as `product` in the preceding example, the framework needs a way to generate those objects. Where exactly do they come from, and how are they created?

I've already hinted that, in most cases, these values come from the request itself. But the HTTP request that the server receives is a series of strings—how does ASP.NET Core turn that into a .NET object? This is where *model binding* comes in.

**DEFINITION** *Model binding* extracts values from a request and uses them to create .NET objects. These objects are passed as method parameters to the page handler being executed or are set as properties of the `PageModel` that are marked with the `[BindProperty]` attribute.

The model binder is responsible for looking through the request that comes in and finding values to use. It then creates objects of the appropriate type and assigns these values to your model in a process called *binding*.

**NOTE** Model binding in Razor Pages and MVC is a one-way population of objects from the request, not the two-way data binding that desktop or mobile development sometimes uses.

Any properties on your Razor Page's `PageModel` (in the `.cshtml.cs` file for your Razor Page), that are decorated with the `[BindProperty]` attribute are created from the incoming request using model binding, as shown in the following listing. Similarly, if your page handler method has any parameters, these are also created using model binding.

#### Listing 6.1 Model binding requests to properties in a Razor Page

```
public class IndexModel: PageModel
{
    [BindProperty]
    public string Category { get; set; }

    [BindProperty(SupportsGet = true)]
    public string Username { get; set; }

    public void OnGet()
    {
    }

    public void OnPost(ProductModel model)
    {
    }
}
```

Properties decorated with `[BindProperty]` take part in model binding.

Properties are not model-bound for GET requests, unless you use `SupportsGet`.

Parameters to page handlers are also model-bound when that handler is selected.

As shown in the preceding listing, `PageModel` properties are *not* model-bound for GET requests, even if you add the `[BindProperty]` attribute. For security reasons, only requests using verbs like POST and PUT are bound. If you *do* want to bind GET requests, you can set the `SupportsGet` property on the `[BindProperty]` attribute to opt in to model binding.

**TIP** To bind `PageModel` properties for GET requests, use the `SupportsGet` property of the attribute, for example, `[BindProperty(SupportsGet = true)]`.

### Which part is the binding model?

Listing 6.1 shows a Razor Page that uses multiple binding models: the `Category` property, the `Username` property, and the `ProductModel` property (in the `OnPost` handler) are all model-bound.

Using multiple models in this way is fine, but I prefer to use an approach that keeps all the model binding in a single, nested class, which I often call `InputModel`. With this approach, the Razor Page in listing 6.1 could be written as follows:

```
public class IndexModel: PageModel
{
    [BindProperty]
    public InputModel Input { get; set; }
    public void OnGet()
    {
    }

    public class InputModel
    {
        public string Category { get; set; }
        public string Username { get; set; }
        public ProductModel Model { get; set; }
    }
}
```

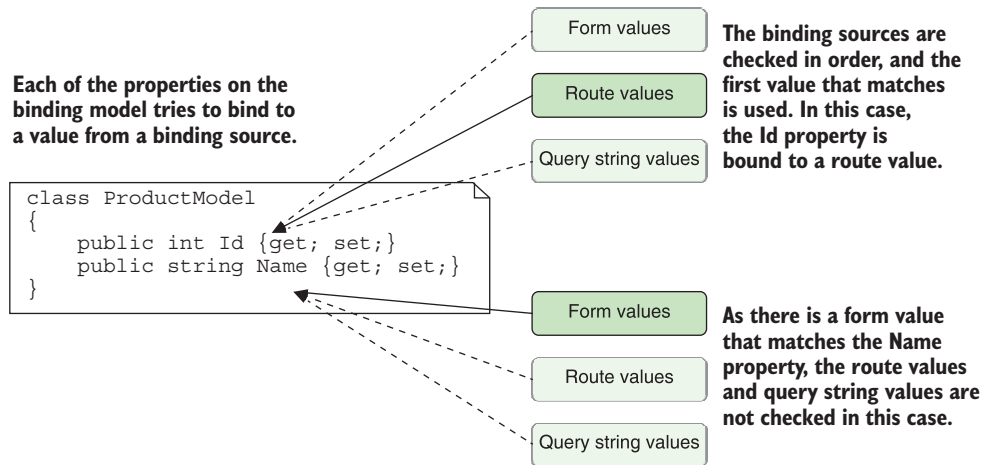
This approach has some organizational benefits that you'll learn more about in section 6.4.

ASP.NET Core automatically populates your binding models for you using properties of the request, such as the request URL, any headers sent in the HTTP request, any data explicitly POSTed in the request body, and so on.

By default, ASP.NET Core uses three different *binding sources* when creating your binding models. It looks through each of these in order and takes the first value it finds (if any) that matches the name of the binding model:

- *Form values*—Sent in the body of an HTTP request when a form is sent to the server using a POST
- *Route values*—Obtained from URL segments or through default values after matching a route, as you saw in chapter 5
- *Query string values*—Passed at the end of the URL, not used during routing

The model binding process is shown in figure 6.3. The model binder checks each binding source to see if it contains a value that could be set on the model. Alternatively, the model can also choose the specific source the value should come from, as you'll see in section 6.2.3. Once each property is bound, the model is validated and is



**Figure 6.3** Model binding involves mapping values from binding sources, which correspond to different parts of a request.

set as a property on the `PageModel` or is passed as a parameter to the page handler. You'll learn about the validation process in the second half of this chapter.

### **PageModel properties or page handler parameters?**

There are two different ways to use model binding in Razor Pages:

- Decorate properties on your `PageModel` with the `[BindProperty]` attribute.
- Add parameters to your page handler method.

Which of these approaches should you choose?

This answer to this question is largely a matter of taste. Setting properties on the `PageModel` and marking them with `[BindProperty]` is the approach you'll see most often in examples. If you use this approach, you'll be able to access the binding model when the view is rendered, as you'll see in chapters 7 and 8.

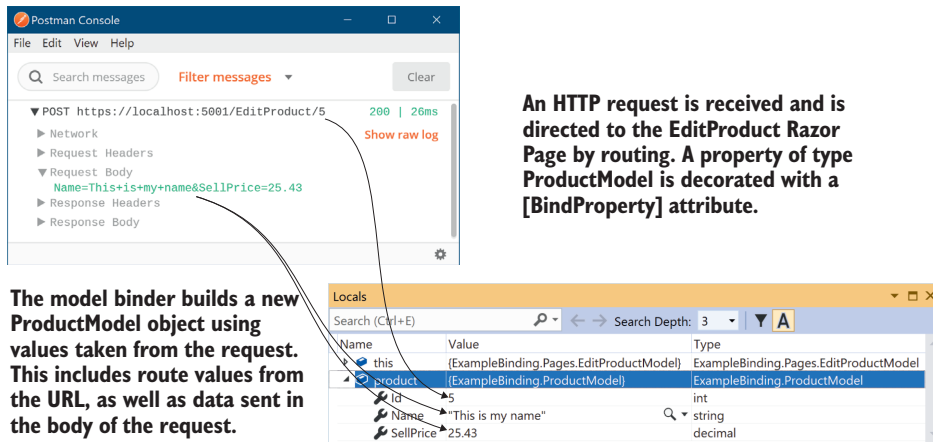
The alternative approach, adding parameters to page handler methods, provides more separation between the different MVC stages, because you won't be able to access the parameters outside of the page handler. On the downside, if you *do* need to display those values in the Razor view, you'll have to manually copy the parameters across to properties that *can* be accessed in the view.

The approach I choose tends to depend on the specific Razor Page I'm building. If I'm creating a form, I will favor the `[BindProperty]` approach, as I typically need access to the request values inside the Razor view. For simple pages, where the binding model is a product ID for example, I tend to favor the page handler parameter approach for its simplicity, especially if the handler is for a `GET` request. I give some more specific advice on my approach in section 6.4.



Figure 6.4 shows an example of a request creating the `ProductModel` method argument using model binding for the example shown at the start of this section:

```
public void OnPost(ProductModel product)
```



**Figure 6.4** Using model binding to create an instance of a model that's used to execute a Razor Page.

The `Id` property has been bound from a URL route parameter, but the `Name` and `SellPrice` properties have been bound from the request body. The big advantage of using model binding is that you don't have to write the code to parse requests and map the data yourself. This sort of code is typically repetitive and error-prone, so using the built-in conventional approach lets you focus on the important aspects of your application: the business requirements.

**TIP** Model binding is great for reducing repetitive code. Take advantage of it whenever possible and you'll rarely find yourself having to access the `Request` object directly.

If you need to, the capabilities are there to let you completely customize the way model binding works, but it's relatively rare that you'll find yourself needing to dig too deep into this. For the majority of cases it works as-is, as you'll see in the remainder of this section.

### 6.2.1 Binding simple types

We'll start our journey into model binding by considering a simple Razor Page handler. The next listing shows a simple Razor Page that takes one number as a method parameter and squares it by multiplying the number by itself.

**Listing 6.2 A Razor Page accepting a simple parameter**

```
public class CalculateSquareModel : PageModel
{
    public void OnGet(int number)
    {
        Square = number * number;
    }

    public int Square { get; set; }
}
```

← The method parameter is the binding model.

← A more complex example would do this work in an external service, in the application model.

← The result is exposed as a property and is used by the view to generate a response.

In the last chapter, you learned about routing and how it selects a Razor Page to execute. You can update the route template for the Razor Page to be "CalculateSquare/{number}" by adding a {number} segment to the Razor Page's @page directive in the .cshtml file, as we discussed in chapter 5:

```
@page "{number}"
```

When a client requests the URL /CalculateSquare/5, the Razor Page framework uses routing to parse it for route parameters. This produces the route value pair:

```
number=5
```

The Razor Page's OnGet page handler contains a single parameter—an integer called *number*—which is your binding model. When ASP.NET Core executes this page handler method, it will spot the expected parameter, flick through the route values associated with the request, and find the *number=5* pair. It can then bind the *number* parameter to this route value and execute the method. The page handler method itself doesn't care about where this value came from; it goes along its merry way, calculating the square of the value, and setting it on the *Square* property.

The key thing to appreciate is that you didn't have to write any extra code to try to extract the number from the URL when the method executed. All you needed to do was create a method parameter (or public property) with the right name and let model binding do its magic.

Route values aren't the only values the model binder can use to create your binding models. As you saw previously, the framework will look through three default *binding sources* to find a match for your binding models:

- Form values
- Route values
- Query string values

Each of these binding sources store values as name-value pairs. If none of the binding sources contain the required value, the binding model is set to a new, default instance

of the type instead. The exact value the binding model will have in this case depends on the type of the variable:

- For value types, the value will be `default(T)`. For an `int` parameter this would be 0, and for a `bool` it would be `false`.
- For reference types, the type is created using the default (parameterless) constructor. For custom types like `ProductModel`, that will create a new object. For nullable types like `int?` or `bool?`, the value will be `null`.
- For string types, the value will be `null`.

**WARNING** It's important to consider the behavior of your page handler when model binding fails to bind your method parameters. If none of the binding sources contain the value, the value passed to the method could be `null` or could unexpectedly have a default value (for value types).

Listing 6.2 showed how to bind a *single* method parameter. Let's take the next logical step and look at how you'd bind *multiple* method parameters.

In the previous chapter, we discussed routing for building a currency converter application. As the next step in your development, your boss asks you to create a method in which the user provides a value in one currency and you must convert it to another. You first create a Razor Page called `Convert.cshtml` and then customize the route template for the page using the `@page` directive to use an absolute path containing two route values:

```
@page "{currencyIn}/{currencyOut}"
```

You then create a page handler that accepts the three values you need, as shown in the following listing.

### Listing 6.3 A Razor Page handler accepting multiple binding parameters

```
public class ConvertModel : PageModel
{
    public void OnGet(
        string currencyIn,
        string currencyOut,
        int qty
    )
    {
        /* method implementation */
    }
}
```

As you can see, there are three different parameters to bind. The question is, where will the values come from and what will they be set to? The answer is, it depends! Table 6.1 shows a whole variety of possibilities. All these examples use the same route template and page handler, but depending on the data sent, different values will be

bound. The actual values might differ from what you expect, as the available binding sources offer conflicting values!

**Table 6.1** Binding request data to page handler parameters from multiple binding sources

URL (route values)	HTTP body data (form values)	Parameter values bound
/GBP/USD		currencyIn=GBP currencyOut=USD qty=0
/GBP/USD?currencyIn=CAD	QTY=50	currencyIn=GBP currencyOut=USD qty=50
/GBP/USD?qty=100	qty=50	currencyIn=GBP currencyOut=USD qty=50
/GBP/USD?qty=100	currencyIn=CAD& currencyOut=EUR&qty=50	currencyIn=CAD currencyOut=EUR qty=50

For each example, be sure you understand *why* the bound values have the values that they do. In the first example, the qty value isn't found in the form data, in the route values, or in the query string, so it has the default value of 0. In each of the other examples, the request contains one or more duplicated values; in these cases, it's important to bear in mind the order in which the model binder consults the binding sources. By default, form values will take precedence over other binding sources, including route values!

**NOTE** The default model binder isn't case sensitive, so a binding value of QTY=50 will happily bind to the qty parameter.

Although this may seem a little overwhelming, it's relatively unusual to be binding from all these different sources at once. It's more common to have your values all come from the request body as form values, maybe with an ID from URL route values. This scenario serves as more of a cautionary tale about the knots you can twist yourself into if you're not sure how things work under the hood.

In these examples, you happily bound the qty integer property to incoming values, but as I mentioned earlier, the values stored in binding sources are all strings. What types can you convert a string to? The model binder will convert pretty much any primitive .NET type such as int, float, decimal (and string obviously), plus anything that has a `TypeConverter`.<sup>2</sup> There are a few other special cases that can be converted from a string, such as `Type`, but thinking of it as primitives only will get you a long way there!

<sup>2</sup> `TypeConverters` can be found in the `System.ComponentModel.TypeConverter` package. You can read more about them in Microsoft's "Type conversion in .NET" documentation: <http://mng.bz/A0GK>.

### 6.2.2 Binding complex types

If it seems like only being able to bind simple primitive types is a bit limiting, then you're right! Luckily, that's not the case for the model binder. Although it can only convert strings *directly* to those primitive types, it's also able to bind complex types by traversing any properties your binding models expose.

If this doesn't make you happy straight off the bat, let's look at how you'd have to build your page handlers if simple types were your only option. Imagine a user of your currency converter application has reached a checkout page and is going to exchange some currency. Great! All you need now is to collect their name, email, and phone number. Unfortunately, your page handler method would have to look something like this:

```
public IActionResult OnPost(string firstName, string lastName, string
    phoneNumber, string email)
```

Yuck! Four parameters might not seem that bad right now, but what happens when the requirements change and you need to collect other details? The method signature will keep growing. The model binder will bind the values quite happily, but it's not exactly clean code. Using the [BindProperty] approach doesn't really help either—you still have to clutter up our PageModel with lots of properties and attributes!

#### SIMPLIFYING METHOD PARAMETERS BY BINDING TO COMPLEX OBJECTS

A common pattern for any C# code when you have many method parameters is to extract a class that encapsulates the data the method requires. If extra parameters need to be added, you can add a new property to this class. This class becomes your binding model, and it might look something like this.

#### Listing 6.4 A binding model for capturing a user's details

```
public class UserBindingModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string PhoneNumber { get; set; }
}
```

With this model, you can now update your page handler's method signature to

```
public IActionResult OnPost(UserBindingModel user)
```

Or alternatively, using the [BindProperty] approach, create a property on the PageModel:

```
[BindProperty]
public UserBindingModel User { get; set; }
```

Now you can simplify the page handler signature even further:

```
public IActionResult OnPost()
```

Functionally, the model binder treats this new complex type a little differently. Rather than looking for parameters with a value that matches the parameter name (`user`, or `User` for the property), the model binder creates a new instance of the model using `new UserBindingModel()`.

**NOTE** You don't have to use custom classes for your methods; it depends on your requirements. If your page handler needs only a single integer, then it makes more sense to bind to the simple parameter.

Next, the model binder loops through all the properties your binding model has, such as `FirstName` and `LastName` in listing 6.4. For each of these properties, it consults the collection of binding sources and attempts to find a name-value pair that matches. If it finds one, it sets the value on the property and moves on to the next.

**TIP** Although the name of the model isn't necessary in this example, the model binder will also look for properties prefixed with the name of the property, such as `user.FirstName` and `user.LastName` for a property called `User`. You can use this approach when you have multiple complex parameters to a page handler, or multiple complex `[BindProperty]` properties. In general, for simplicity, you should avoid this situation if possible. As for all model binding, the casing of the prefix does not matter.

Once all the properties that can be bound on the binding model are set, the model is passed to the page handler (or the `[BindProperty]` property is set), and the handler is executed as usual. The behavior from this point on is identical to when you have lots of individual parameters—you'll end up with the same values set on your binding model—but the code is cleaner and easier to work with.

**TIP** For a class to be model-bound, it must have a default public constructor. You can only bind properties that are public and settable.

With this technique you can bind complex hierarchical models whose properties are *themselves* complex models. As long as each property exposes a type that can be model-bound, the binder can traverse it with ease.

#### BINDING COLLECTIONS AND DICTIONARIES

As well as ordinary custom classes and primitives, you can bind to collections, lists, and dictionaries. Imagine you had a page in which a user selected all the currencies they were interested in; you'd display the rates for all those selected, as shown in figure 6.5.

To achieve this, you could create a page handler that accepts a `List<string>` type, such as

```
public void OnPost(List<string> currencies);
```

Selected Currencies	Exchange Rates
GBP <b>USD</b> CAD EUR	<b>USD</b> 1.22 <b>CAD</b> 1.64

**Figure 6.5** The select list in the currency converter application will send a list of selected currencies to the application. Model binding can bind the selected currencies and customize the view for the user to show the equivalent cost in the selected currencies.

You could then POST data to this method by providing values in several different formats:

- `currencies[index]`—Where `currencies` is the name of the parameter to bind and `index` is the index of the item to bind, for example, `currencies[0]=GBP&currencies[1]=USD`.
- `[index]`—If you're only binding to a single list (as in this example), you can omit the name of the parameter, for example, `[0]=GBP&[1]=USD`.
- `currencies`—Alternatively, you can omit the index and send `currencies` as the key for every value, for example, `currencies=GBP&currencies=USD`.

The key values can come from route values and query values, but it's far more common to POST them in a form. Dictionaries can use similar binding, where the dictionary key replaces the index both when the parameter is named and when it's omitted.

If this all seems a bit confusing, don't feel too alarmed. If you're building a traditional web application and using Razor views to generate HTML, the framework will take care of generating the correct names for you. As you'll see in chapter 8, the Razor view will ensure that any form data you POST will be generated in the correct format.

#### **BINDING FILE UPLOADS WITH IFORMFILE**

A common feature of many websites is the ability to upload files. This could be a relatively infrequent activity, such as a user uploading a profile picture for their Stack Overflow profile, or it may be integral to the application, like uploading photos to Facebook.

### Letting users upload files to your application

Uploading files to websites is a pretty common activity, but you should carefully consider whether your application *needs* that ability. Whenever files can be uploaded by users, the road is fraught with danger.

You should be careful to treat the incoming files as potentially malicious: don't trust the filename provided, take care of large files being uploaded, and don't allow the files to be executed on your server.

Files also raise questions as to where the data should be stored—should they go in a database, in the filesystem, or in some other storage? None of these questions has a straightforward answer, and you should think hard about the implications of choosing one over the other. Better yet, if you can avoid it, don't let users upload files!

ASP.NET Core supports uploading files by exposing the `IFormFile` interface. You can use this interface as your binding model, either as a method parameter to your page handler, or using the `[BindProperty]` approach, and it will be populated with the details of the file upload:

```
public void OnPost(IFormFile file);
```

You can also use an `IEnumerable<IFormFile>` if you need to accept multiple files:

```
public void OnPost(IEnumerable<IFormFile> file);
```

The `IFormFile` object exposes several properties and utility methods for reading the contents of the uploaded file, some of which are shown here:

```
public interface IFormFile
{
    string ContentType { get; }
    long Length { get; }
    string FileName { get; }
    Stream OpenReadStream();
}
```

As you can see, this interface exposes a `FileName` property, which returns the filename that the file was uploaded with. But you know not to trust users, right? You should *never* use the filename directly in your code—always generate a new filename for the file before you save it anywhere.

**WARNING** Never use posted filenames in your code. Users can use them to attack your website and access files they shouldn't be able to.

The `IFormFile` approach is fine if users are only going to be uploading small files. When your method accepts an `IFormFile` instance, the whole content of the file is buffered in memory and on disk before you receive it. You can then use the `OpenReadStream` method to read the data out.



If users post large files to your website, you may find you start to run out of space in memory or on disk, as it buffers each of the files. In that case, you may need to stream the files directly to avoid saving all the data at once. Unfortunately, unlike the model-binding approach, streaming large files can be complex and error-prone, so it's outside the scope of this book. For details, see Microsoft's "Upload files in ASP.NET Core" documentation at <http://mng.bz/S7X>.

**TIP** Don't use the `IFormFile` interface to handle large file uploads as you may see performance issues. Be aware that you can't rely on users *not* to upload large files, so better yet, avoid file uploads entirely!

For the vast majority of Razor Pages, the default configuration of model binding for simple and complex types works perfectly well, but you may find some situations where you need to take a bit more control. Luckily, that's perfectly possible, and you can completely override the process if necessary by replacing the `ModelBinders` used in the guts of the framework.

However, it's rare to need that level of customization—I've found it's more common to want to specify which *binding source* to use for a page's binding model instead.

### 6.2.3 Choosing a binding source

As you've already seen, by default the ASP.NET Core model binder will attempt to bind your binding models from three different binding sources: form data, route data, and the query string.

Occasionally, you may find it necessary to specifically declare which binding source to bind to. In other cases, these three sources won't be sufficient at all. The most common scenarios are when you want to bind a method parameter to a request header value, or when the body of a request contains JSON-formatted data that you want to bind to a parameter. In these cases, you can decorate your binding models with attributes that say where to bind from, as shown in the following listing.

#### Listing 6.5 Choosing a binding source for model binding

```
public class PhotosModel: PageModel
{
    public void OnPost(
        [FromHeader] string userId,
        [FromBody] List<Photo> photos)
    {
        /* method implementation */
    }
}
```

The `userId` will be bound from an HTTP header in the request.

The list of photo objects will be bound to the body of the request, typically in JSON format.

In this example, a page handler updates a collection of photos with a user ID. There are method parameters for the ID of the user to be tagged in the photos, `userId`, and a list of `Photo` objects to tag, `photos`.

Rather than binding these method parameters using the standard binding sources, I've added attributes to each parameter, indicating the binding source to use. The

[FromHeader] attribute has been applied to the `userId` parameter. This tells the model binder to bind the value to an HTTP request header value called `userId`.

We're also binding a list of photos to the body of the HTTP request by using the [FromBody] attribute. This will read JSON from the body of the request and will bind it to the `List<Photo>` method parameter.

**WARNING** Developers familiar with the previous version of ASP.NET should take note that the [FromBody] attribute is explicitly required when binding to JSON requests in Razor Pages. This differs from previous ASP.NET behavior, in which no attribute was required.

You aren't limited to binding JSON data from the request body—you can use other formats too, depending on which `InputFormatters` you configure the framework to use. By default, only a JSON input formatter is configured. You'll see how to add an XML formatter in chapter 9, when I discuss Web APIs.

You can use a few different attributes to override the defaults and to specify a binding source for each binding model (or each property on the binding model):

- [FromHeader]—Bind to a header value
- [FromQuery]—Bind to a query string value
- [FromRoute]—Bind to route parameters
- [FromForm]—Bind to form data posted in the body of the request
- [FromBody]—Bind to the request's body content

You can apply each of these to any number of handler method parameters or properties, as you saw in listing 6.5, with the exception of the [FromBody] attribute—only one value may be decorated with the [FromBody] attribute. Also, as form data is sent in the body of a request, the [FromBody] and [FromForm] attributes are effectively mutually exclusive.

**TIP** Only one parameter may use the [FromBody] attribute. This attribute will consume the incoming request as HTTP request bodies can only be safely read once.

As well as these attributes for specifying binding sources, there are a few other attributes for customizing the binding process even further:

- [BindNever]—The model binder will skip this parameter completely.<sup>3</sup>
- [BindRequired]—If the parameter was not provided, or was empty, the binder will add a validation error.
- [FromServices]—This is used to indicate the parameter should be provided using dependency injection (see chapter 10 for details).

---

<sup>3</sup> You can use the [BindNever] attribute to prevent mass assignment, as discussed in these two posts on my blog: <http://mng.bz/QyfG> and <http://mng.bz/Vd90>.

In addition, you have the `[ModelBinder]` attribute, which puts you into “God mode” with respect to model binding. With this attribute, you can specify the exact binding source, override the name of the parameter to bind to, and specify the type of binding to perform. It’ll be rare that you need this one, but when you do, at least it’s there!

By combining all these attributes, you should find you’re able to configure the model binder to bind to pretty much any request data your page handler wants to use. In general, though, you’ll probably find you rarely need to use them; the defaults should work well for you in most cases.

That brings us to the end of this section on model binding. If all has gone well, your page handler should have access to a populated binding model, and it’s ready to execute its logic. It’s time to handle the request, right? Nothing to worry about?

Not so fast! How do you know that the data you received was valid? That you haven’t been sent malicious data attempting a SQL injection attack, or a phone number full of letters?

The binder is relatively blindly assigning values sent in a request, which you’re happily going to plug into your own methods? What’s to stop nefarious little Jimmy from sending malicious values to your application?

Except for basic safeguards, there’s nothing stopping him, which is why it’s important that you *always* validate the input coming in. ASP.NET Core provides a way to do this in a declarative manner out of the box, which is the focus of the second half of this chapter.

## 6.3 *Handling user input with model validation*

In this section I discuss

- What is validation, and why do you need it?
- Using `DataAnnotations` attributes to describe the data you expect
- How to validate your binding models in page handlers

Validation in general is a pretty big topic, and it’s one that you’ll need to consider in every app you build. ASP.NET Core makes it relatively easy to add validation to your applications by making it an integral part of the framework.

### 6.3.1 *The need for validation*

Data can come from many different sources in your web application—you could load it from files, read it from a database, or accept values that a user typed into a form in requests. Although you might be inclined to trust that the data already on your server is valid (though this is sometimes a dangerous assumption!), you *definitely* shouldn’t trust the data sent as part of a request.

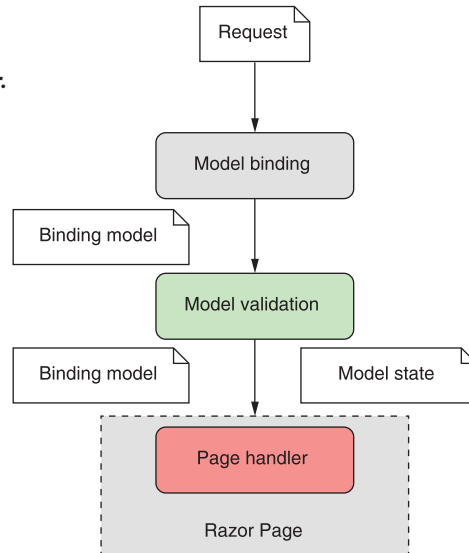
Validation occurs in the Razor Pages framework after model binding, but before the page handler executes, as you saw in figure 6.2. Figure 6.6 shows a more compact view of where model validation fits in this process, demonstrating how a request to a checkout page that requests a user’s personal details is bound and validated.

1. A request is received for the URL `/checkout/saveuser`, and the routing middleware selects the `SaveUser` Razor Page endpoint in the `Checkout` folder.

2. The framework builds a `UserBindingModel` from the details provided in the request.

3. The `UserBindingModel` is validated according to the `DataAnnotations` attributes on its properties.

4. The `UserBindingModel` and validation `ModelState` are set on the `SaveUser` Razor Page, and the page handler is executed.



**Figure 6.6** Validation occurs after model binding but before the page handler executes. The page handler executes whether or not validation is successful.

*You should always validate data provided by users before you use it in your methods.* You have no idea what the browser may have sent you. The classic example of “little Bobby Tables” (<https://xkcd.com/327/>) highlights the need to always validate any data sent by a user.

Validation isn’t only to check for security threats, though; it’s also needed to check for non-malicious errors:

- Data should be formatted correctly (email fields have a valid email format).
- Numbers might need to be in a particular range (you can’t buy -1 copies of this book!).
- Some values may be required but others are optional (name may be required for a profile but phone number is optional).
- Values must conform to your business requirements (you can’t convert a currency to itself, it needs to be converted to a different currency).

It might seem like some of these can be dealt with easily enough in the browser. For example, if a user is selecting a currency to convert to, don’t let them pick the same currency; and we’ve all seen the “please enter a valid email address” messages.

Unfortunately, although this *client-side validation* is useful for users, as it gives them instant feedback, you can never rely on it, as it will *always* be possible to bypass these browser protections. It’s always necessary to validate the data as it arrives at your web application using *server-side validation*.

**WARNING** Always validate user input on the server side of your application.

If that feels a little redundant, like you'll be duplicating logic and code, then I'm afraid you're right. It's one of the unfortunate aspects of web development; the duplication is a necessary evil. Thankfully, ASP.NET Core provides several features to try to reduce this burden.

**TIP** Blazor, the new C# SPA framework promises to solve some of these issues. For details, see <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor> and *Blazor in Action* by Chris Sainty (Manning, 2021).

If you had to write this validation code fresh for every app, it would be tedious and likely error prone. Luckily, you can simplify your validation code significantly using a set of attributes provided by .NET Core and .NET 5.0.

### 6.3.2 Using DataAnnotations attributes for validation

Validation attributes, or more precisely DataAnnotations attributes, allow you to specify the rules that your binding model should conform to. They provide *metadata* about your model by describing the *sort* of data the binding model should contain, as opposed to the data itself.

**DEFINITION** *Metadata* describes other data, specifying the rules and characteristics the data should adhere to.

You can apply DataAnnotations attributes directly to your binding models to indicate the type of data that's acceptable. This allows you to, for example, check that required fields have been provided, that numbers are in the correct range, and that email fields are valid email addresses.

As an example, let's consider the checkout page for your currency converter application. You need to collect details about the user before you can continue, so you ask them to provide their name, email, and, optionally, a phone number. The following listing shows the `UserBindingModel` decorated with validation attributes that represent the validation rules for the model. This expands on the example you saw in listing 6.4.

**Listing 6.6** Adding DataAnnotations to a binding model to provide metadata

```
public class UserBindingModel
{
    [Required]
    [StringLength(100)]
    [Display(Name = "Your name")]
    public string FirstName { get; set; }

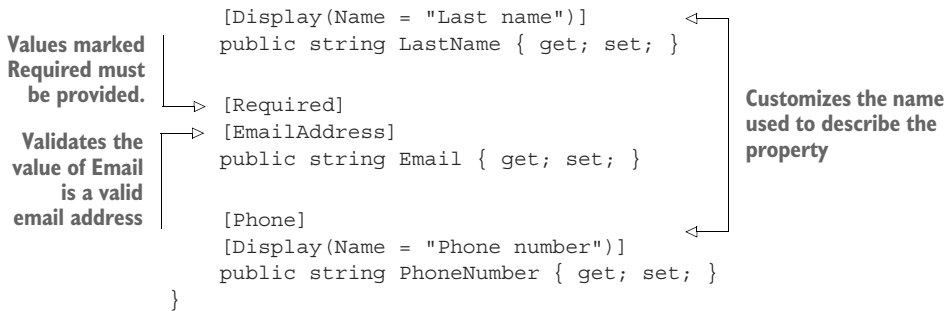
    [Required]
    [StringLength(100)]
    public string LastName { get; set; }
}
```

Values marked Required must be provided.

The `StringLengthAttribute` sets the maximum length for the property.

Customizes the name used to describe the property

The `StringLengthAttribute` sets the maximum length for the property.



Suddenly your binding model contains a whole wealth of information where previously it was pretty sparse on details. For example, you've specified that the `FirstName` property should always be provided, that it should have a maximum length of 100 characters, and that when it's referred to (in error messages, for example) it should be called "Your name" instead of "FirstName".

The great thing about these attributes is that they clearly declare the *expected* state of the model. By looking at these attributes, you know what the properties will, or should, contain. They also provide hooks for the ASP.NET Core framework to validate that the data set on the model during model binding is valid, as you'll see shortly.

You've got a plethora of attributes to choose from when applying `DataAnnotations` to your models. I've listed some of the common ones here, but you can find more in the `System.ComponentModel.DataAnnotations` namespace. For a more complete list, I recommend using IntelliSense in Visual Studio/Visual Studio Code, or you can always look at the source code directly on GitHub (<http://mng.bz/5jxZ>).

- `[CreditCard]`—Validates that a property has a valid credit card format.
- `[EmailAddress]`—Validates that a property has a valid email address format.
- `[StringLength(max)]`—Validates that a string has at most max number of characters.
- `[MinLength(min)]`—Validates that a collection has at least the min number of items.
- `[Phone]`—Validates that a property has a valid phone number format.
- `[Range(min, max)]`—Validates that a property has a value between min and max.
- `[RegularExpression(regex)]`—Validates that a property conforms to the regex regular expression pattern.
- `[Url]`—Validates that a property has a valid URL format.
- `[Required]`—Indicates the property must not be null.
- `[Compare]`—Allows you to confirm that two properties have the same value (for example, `Email` and `ConfirmEmail`).

**WARNING** The [EmailAddress] and other attributes only validate that the *format* of the value is correct. They don't validate that the email address exists.<sup>4</sup>

The DataAnnotations attributes aren't a new feature—they have been part of the .NET Framework since version 3.5—and their usage in ASP.NET Core is almost the same as in the previous version of ASP.NET.

They're also used for other purposes, in addition to validation. Entity Framework Core (among others) uses DataAnnotations to define the types of columns and rules to use when creating database tables from C# classes. You can read more about Entity Framework Core in chapter 12, and in Jon P. Smith's *Entity Framework Core in Action*, second edition (Manning, 2021).

If the DataAnnotation attributes provided out of the box don't cover everything you need, it's also possible to write custom attributes by deriving from the base `ValidationAttribute`. You'll see how to create a custom attribute for your currency converter application in chapter 19.

Alternatively, if you're not a fan of the attribute-based approach, ASP.NET Core is flexible enough that you can completely replace the validation infrastructure with your preferred technique. For example, you could use the popular FluentValidation library (<https://github.com/JeremySkinner/FluentValidation>) in place of the DataAnnotations attributes if you prefer. You'll see how to do this in chapter 20.

**TIP** DataAnnotations are good for input validation of properties in isolation, but not so good for validating business rules. You'll most likely need to perform this validation outside the DataAnnotations framework.

Whichever validation approach you use, it's important to remember that these techniques don't protect your application by themselves. The Razor Pages framework will ensure that validation occurs, but it doesn't automatically do anything if validation fails. In the next section we'll look at how to check the validation result on the server and handle the case where validation has failed.

### 6.3.3 Validating on the server for safety

Validation of the binding model occurs before the page handler executes, but note that the handler *always* executes, whether the validation failed or succeeded. It's the responsibility of the page handler to check the result of the validation.

**NOTE** Validation happens automatically, but handling validation failures is the responsibility of the page handler.

The Razor Pages framework stores the output of the validation attempt in a property on the `PageModel` called `ModelState`. This property is a `ModelStateDictionary` object,

---

<sup>4</sup> The phone number attribute is particularly lenient in the formats it allows. For an example of this, and how to do more rigorous phone number validation, see this post on the Twilio blog: <http://mng.bz/xmZe>.

which contains a list of all the validation errors that occurred after model binding, as well as some utility properties for working with it.

As an example, the following listing shows the `OnPost` page handler for the `Checkout.cshtml` Razor Page. The `Input` property is marked for binding and uses the `UserBindingModel` type shown previously in listing 6.6. This page handler doesn't do anything with the data currently, but the pattern of checking `ModelState` early in the method is the key takeaway here.

#### Listing 6.7 Checking model state to view the validation result

```
public class CheckoutModel : PageModel
{
    [BindProperty]
    public UserBindingModel Input { get; set; }

    public IActionResult OnPost()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        /* Save to the database, update user, return success */

        return RedirectToPage("Success");
    }
}
```

The `ModelState` property is available on the `PageModel` base class.

The `Input` property contains the model-bound data.

The binding model is validated before the page handler is executed.

If there were validation errors, `IsValid` will be false.

Validation failed, so redisplay the form with errors and finish the method early.

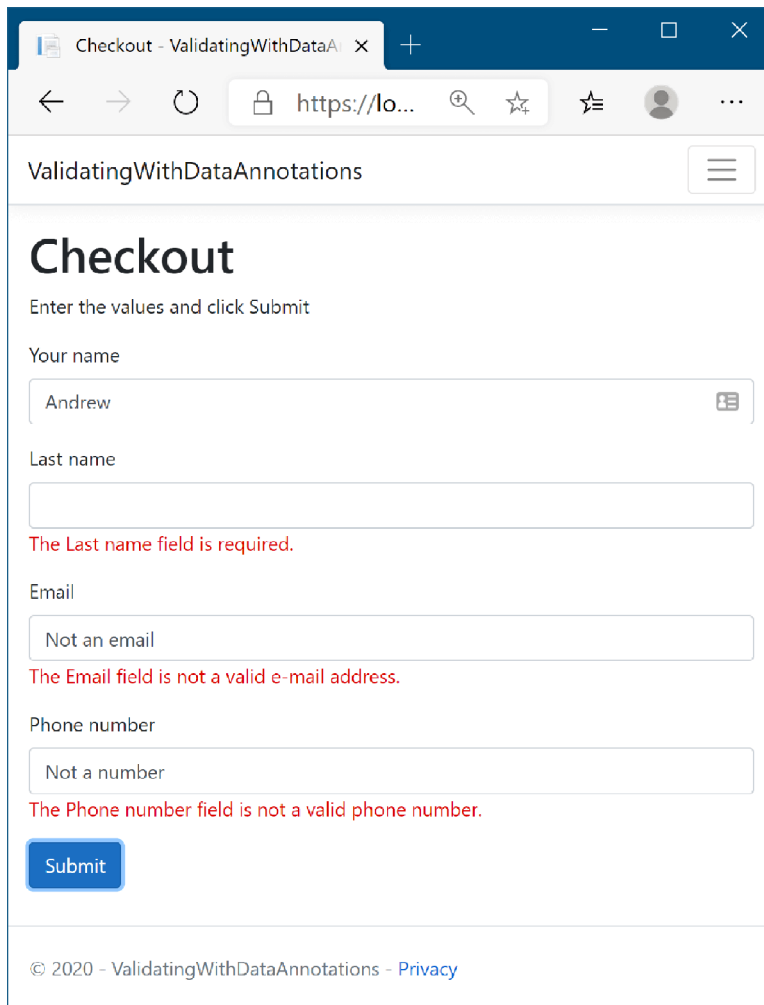
Validation passed, so it's safe to use the data provided in the model.

If the `ModelState` property indicates an error occurred, the method immediately calls the `Page` helper method. This returns a `PageResult` that will ultimately generate HTML to return to the user, as you saw in chapter 4. The view uses the (invalid) values provided in the `Input` property to repopulate the form when it's displayed, as shown in figure 6.7. Also, helpful messages for the user are added automatically, using the validation errors in the `ModelState` property.

**NOTE** The error messages displayed on the form are the default values for each validation attribute. You can customize the message by setting the `ErrorMessage` property on any of the validation attributes. For example, you could customize a `[Required]` attribute using `[Required(ErrorMessage="Required")]`.

If the request is successful, the page handler returns a `RedirectToPageResult` (using the `RedirectToPage()` helper method) that redirects the user to the `Success.cshtml` Razor Page. This pattern of returning a redirect response after a successful POST is called the POST-REDIRECT-GET pattern.





The screenshot shows a web browser window with the title "Checkout - ValidatingWithDataAnnotations". The address bar shows "https://lo...". The page content includes a header "ValidatingWithDataAnnotations" and a main heading "Checkout". Below the heading is a subheading "Enter the values and click Submit". The form contains four input fields: "Your name" (with the value "Andrew"), "Last name" (empty), "Email" (with the value "Not an email"), and "Phone number" (with the value "Not a number"). Below the "Last name" field is a red error message: "The Last name field is required." Below the "Email" field is a red error message: "The Email field is not a valid e-mail address." Below the "Phone number" field is a red error message: "The Phone number field is not a valid phone number." A blue "Submit" button is located below the "Phone number" field. At the bottom of the page is a footer: "© 2020 - ValidatingWithDataAnnotations - [Privacy](#)".

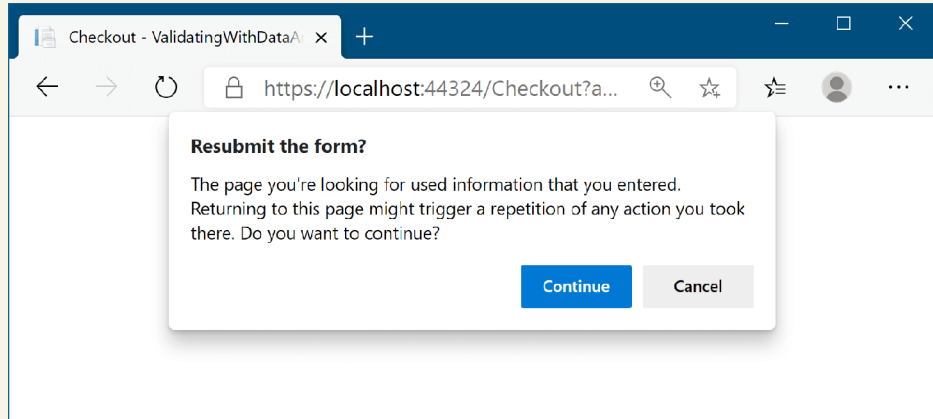
**Figure 6.7** When validation fails, you can redisplay the form to display `ModelState` validation errors to the user. Note the Your Name field has no associated validation errors, unlike the other fields.

### POST-REDIRECT-GET

The POST-REDIRECT-GET design pattern is a web development pattern that prevents users from accidentally submitting the same form multiple times. Users typically submit a form using the standard browser `POST` mechanism, sending data to the server. This is the normal way by which you might take a payment, for example.

If a server takes the naive approach and responds with a `200 OK` response and some HTML to display, the user will still be on the same URL. If the user then refreshes

their browser, they will be making an *additional* POST to the server, potentially making *another* payment! Browsers have some mechanisms to avoid this, such as in the following figure, but the user experience isn't desirable.



**Refreshing a browser window after a POST causes a warning message to be shown to the user.**

The POST-REDIRECT-GET pattern says that in response to a successful POST, you should return a REDIRECT response to a new URL, which will be followed by the browser making a GET to the new URL. If the user refreshes their browser now, they'll be refreshing the final GET call to the new URL. No additional POST is made, so no additional payments or side effects should occur.

This pattern is easy to achieve in ASP.NET Core MVC applications using the pattern shown in listing 6.7. By returning a `RedirectToPageResult` after a successful POST, your application will be safe if the user refreshes the page in their browser.

You might be wondering why ASP.NET Core doesn't handle invalid requests for you automatically—if validation has failed, and you have the result, why does the page handler get executed at all? Isn't there a risk that you might forget to check the validation result?

This is true, and in some cases the best thing to do is to make the generation of the validation check and response automatic. In fact, this is exactly the approach we will use for Web APIs when we cover them in chapter 9.

For Razor Pages apps however, you typically still want to generate an HTML response, even when validation failed. This allows the user to see the problem and potentially correct it. This is much harder to make automatic.

For example, you might find you need to load additional data before you can redisplay the Razor Page—such as loading a list of available currencies. That becomes

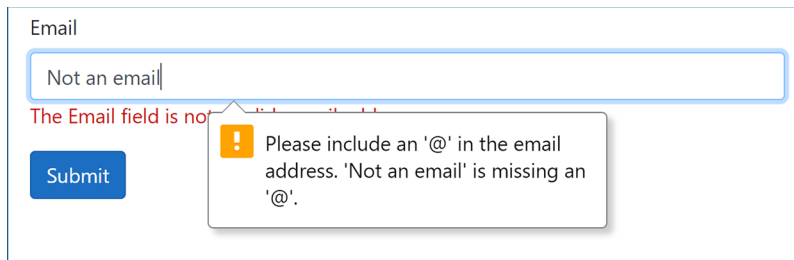
simpler and more explicit with the `IsValid` pattern. Trying to do that automatically would likely end up with you fighting against edge cases and workarounds.

Also, by including the `IsValid` check explicitly in your page handlers, it's easier to control what happens when *additional* validation checks fail. For example, if the user tries to update a product, the `DataAnnotations` validation won't know whether a product with the requested ID exists, only whether the ID has the correct *format*. By moving the validation to the handler method, you can treat data and business rule validation failures in the same way.

I hope I've hammered home how important it is to validate user input in ASP.NET Core, but just in case: **VALIDATE!** There, we're good. Having said that, *only* performing validation on the server can leave users with a slightly poor experience. How many times have you filled out a form online, submitted it, gone to get a snack, and come back to find out you mistyped something and have to redo it. Wouldn't it be nicer to have that feedback immediately?

### 6.3.4 Validating on the client for user experience

You can add client-side validation to your application in a couple of different ways. HTML5 has several built-in validation behaviors that many browsers will use. If you display an email address field on a page and use the "email" HTML input type, the browser will automatically stop you from submitting an invalid format, as shown in figure 6.8.



**Figure 6.8** By default, modern browsers will automatically validate fields of the email type before a form is submitted.

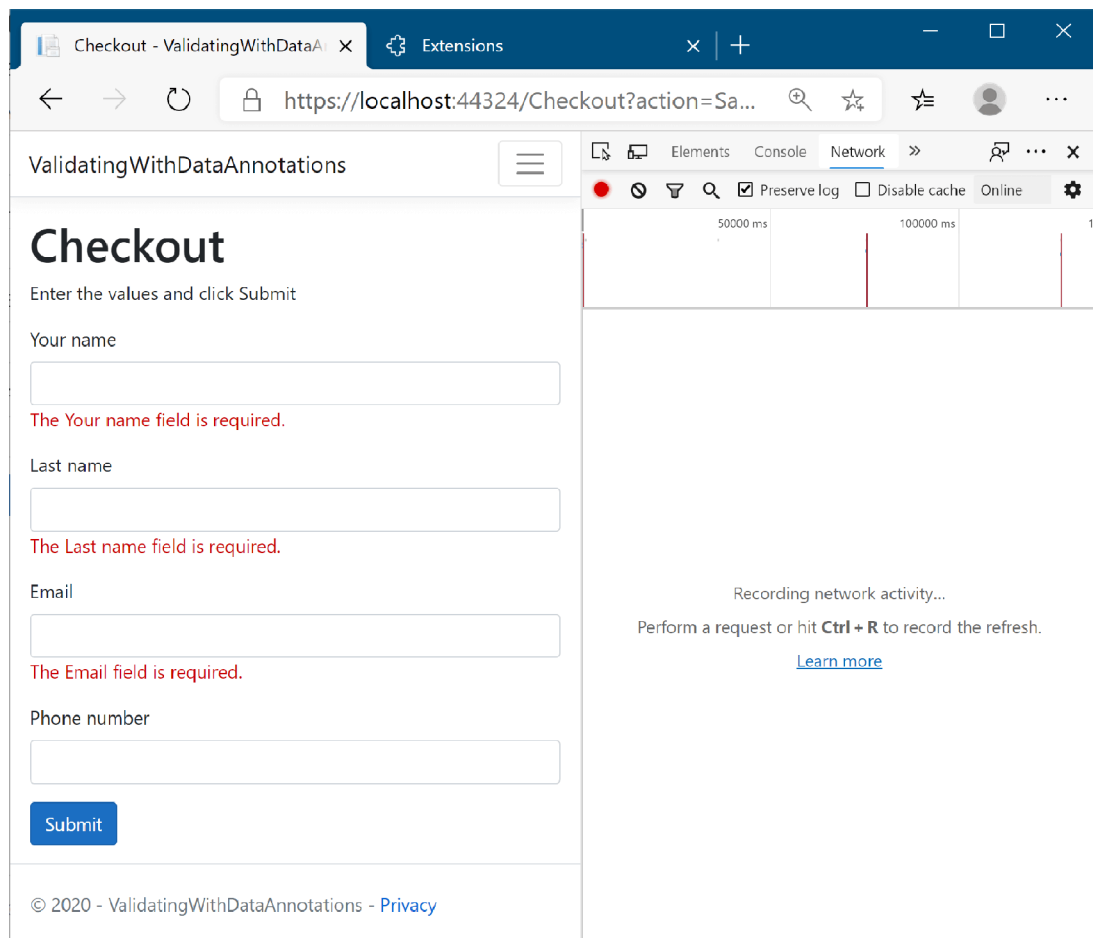
Your application doesn't control this validation; it's built into modern HTML5 browsers.<sup>5</sup> The alternative approach is to perform client-side validation by running JavaScript on the page and checking the values the user has entered before submitting the form. This is the most common approach used in Razor Pages.

<sup>5</sup> HTML5 constraint validation support varies by browser. For details on the available constraints, see the Mozilla documentation (<http://mng.bz/daX3>) and <https://caniuse.com/#feat=constraint-validation>.

I'll go into detail on how to generate the client-side validation helpers in the next chapter, where you'll see the `DataAnnotations` attributes come to the fore once again. By decorating a view model with these attributes, you provide the necessary metadata to the Razor engine for it to generate the appropriate HTML.

With this approach, the user sees any errors with their form immediately, even before the request is sent to the server, as shown in figure 6.9. This gives a much shorter feedback cycle, providing a much better user experience.

If you're building an SPA, the onus is on the client-side framework to validate the data on the client side before posting it to the Web API. The Web API will still validate the data when it arrives at the server, but the client-side framework is responsible for providing the smooth user experience.



**Figure 6.9** With client-side validation, clicking **Submit** will trigger validation to be shown in the browser before the request is sent to the server. As shown in the right pane, no request is sent.

When you use Razor Pages to generate your HTML, you get much of this validation for free. It automatically configures client-side validation for most of the built-in attributes without requiring additional work, as you'll see in chapter 7. Unfortunately, if you've used custom `ValidationAttributes`, these will only run on the server by default; you need to do some additional wiring up of the attribute to make it work on the client side too. Despite this, custom validation attributes can be useful for handling common validation scenarios in your application, as you'll see in chapter 20.

The model binding framework in ASP.NET Core gives you a lot of options on how to organize your Razor Pages: page handler parameters or `PageModel` properties; one binding model or multiple; options for where to define your binding model classes. In the next section I give some advice on how *I* like to organize my Razor Pages.

## 6.4 Organizing your binding models in Razor Pages

In this section I give some general advice on how I like to configure the binding models in my Razor Pages. If you follow the patterns in this section, your Razor Pages will follow a consistent layout, making it easier for others to understand how each Razor Page in your app works.

**NOTE** This advice is just personal preference, so feel free to adapt it if there are aspects you don't agree with. The important thing is to understand *why* I make each suggestion, and to take that on board. Where appropriate, I deviate from these guidelines too!

Model binding in ASP.NET Core has a lot of equivalent approaches to take, so there is no “correct” way to do it. The following listing shows an example of how I would design a simple Razor Page. This Razor Page displays a form for a product with a given ID and allows you to edit the details using a POST request. It's a much longer sample than we've looked at so far, but I highlight the important points below.

### Listing 6.8 Designing an edit product Razor Page

```
public class EditProductModel : PageModel
{
    private readonly ProductService _productService;
    public EditProductModel(ProductService productService)
    {
        _productService = productService;
    }

    [BindProperty]
    public InputModel Input { get; set; }

    public IActionResult OnGet(int id)
    {
        var product = _productService.GetProduct(id);
    }
}
```

**The ProductService is injected using DI and provides access to the application model.**

**A single property is marked with BindProperty.**

**The id parameter is model-bound from the route template for both OnGet and OnPost handlers.**

**Load the product details from the application model.**

```

        Input = new InputModel
        {
            Name = product.ProductName,
            Price = product.SellPrice,
        };
        return Page();
    }

    public IActionResult OnPost(int id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _productService.UpdateProduct(id, Input.Name, Input.Price);

        return RedirectToPage("Index");
    }

    public class InputModel
    {
        [Required]
        public string Name { get; set; }

        [Range(0, int.MaxValue)]
        public decimal Price { get; set; }
    }
}

```

**If the request was not valid, redisplay the form without saving.**

**Build an instance of the InputModel for editing in the form from the existing product's details.**

**The id parameter is model-bound from the route template for both OnGet and OnPost handlers.**

**Redirect to a new page using the POST-REDIRECT-GET pattern.**

**Update the product in the application model using the ProductService.**

**Define the InputModel as a nested class in the Razor Page.**

This page shows the PageModel for a typical “edit form.” These are very common in many line-of-business applications, among others, and it’s a scenario that Razor Pages works very well for. You’ll see how to create the HTML side of forms in chapter 8.

**NOTE** The purpose of this example is only to highlight the model-binding approach. The code is overly simplistic from a *logic* point of view. For example, it doesn’t check that the product with the provided ID exists, or include any error handling.

This form shows several patterns related to model binding that I try to adhere to when building Razor Pages:

- *Only bind a single property with [BindProperty].* I favor having a single property decorated with [BindProperty] for model binding in general. When more than one value needs to be bound, I create a separate class, InputModel, to hold the values, and I decorate that single property with [BindProperty]. Decorating a single property like this makes it harder to forget to add the attribute, and it means all of your Razor Pages use the same pattern.
- *Define your binding model as a nested class.* I define the InputModel as a nested class inside my Razor Page. The binding model is normally highly specific to

that single page, so doing this keeps everything you're working on together. Additionally, I normally use that exact class name, `InputModel`, for all my pages. Again, this adds consistency to your Razor Pages.

- *Don't use `[BindProperties]`.* In addition to the `[BindProperty]` attribute, there is a `[BindProperties]` attribute (note the different spelling) that can be applied to the Razor Page `PageModel` directly. This will cause *all* properties in your model to be model-bound, which can leave you open to over-posting attacks if you're not careful. I suggest you don't use the `[BindProperties]` attribute and stick to binding a *single* property with `[BindProperty]` instead.
- *Accept route parameters in the page handler.* For simple route parameters, such as the `id` passed into the `OnGet` and `OnPost` handlers in listing 6.8, I add parameters to the page handler method itself. This avoids the clunky `SupportsGet=true` syntax for GET requests.
- *Always validate before using data.* I said it before, so I'll say it again. Validate user input!

That concludes this look at model-binding in Razor Pages. You saw how the ASP.NET Core framework uses model binding to simplify the process of extracting values from a request and turning them into normal .NET objects you can quickly work with. The most important aspect of this chapter is the focus on validation—this is a common concern for all web applications, and the use of `DataAnnotations` can make it easy to add validation to your models.

In the next chapter, we'll continue our journey through Razor Pages by looking at how to create views. In particular, you'll learn how to generate HTML in response to a request using the Razor templating engine.

## Summary

- Razor Pages uses three distinct models, each responsible for a different aspect of a request. The binding model encapsulates data sent as part of a request. The application model represents the state of the application. The `PageModel` is the backing class for the Razor Page, and it exposes the data used by the Razor view to generate a response.
- Model binding extracts values from a request and uses them to create .NET objects the page handler can use when they execute.
- Any properties on the `PageModel` marked with the `[BindProperty]` attribute, and method parameters of the page handlers, will take part in model binding.
- Properties decorated with `[BindProperty]` are not bound for GET requests. To bind GET requests, you must use `[BindProperty(SupportsGet = true)]` instead.
- By default, there are three binding sources: POSTed form values, route values, and the query string. The binder will interrogate these in order when trying to bind your binding models.

- When binding values to models, the names of the parameters and properties aren't case sensitive.
- You can bind to simple types or to the properties of complex types.
- To bind complex types, they must have a default constructor and public, settable properties.
- Simple types must be convertible to strings to be bound automatically; for example, numbers, dates, and Boolean values.
- Collections and dictionaries can be bound using the `[index]=value` and `[key]=value` syntax, respectively.
- You can customize the binding source for a binding model using `[From*]` attributes applied to the method, such as `[FromHeader]` and `[FromBody]`. These can be used to bind to nondefault binding sources, such as headers or JSON body content.
- In contrast to the previous version of ASP.NET, the `[FromBody]` attribute is required when binding JSON properties (previously it was not required).
- Validation is necessary to check for security threats. Check that data is formatted correctly and confirm that it conforms to expected values and that it meets your business rules.
- ASP.NET Core provides `DataAnnotations` attributes to allow you to declaratively define the expected values.
- Validation occurs automatically after model binding, but you must manually check the result of the validation and act accordingly in your page handler by interrogating the `ModelState` property.
- Client-side validation provides a better user experience than server-side validation alone, but you should always use server-side validation.
- Client-side validation uses JavaScript and attributes applied to your HTML elements to validate form values.